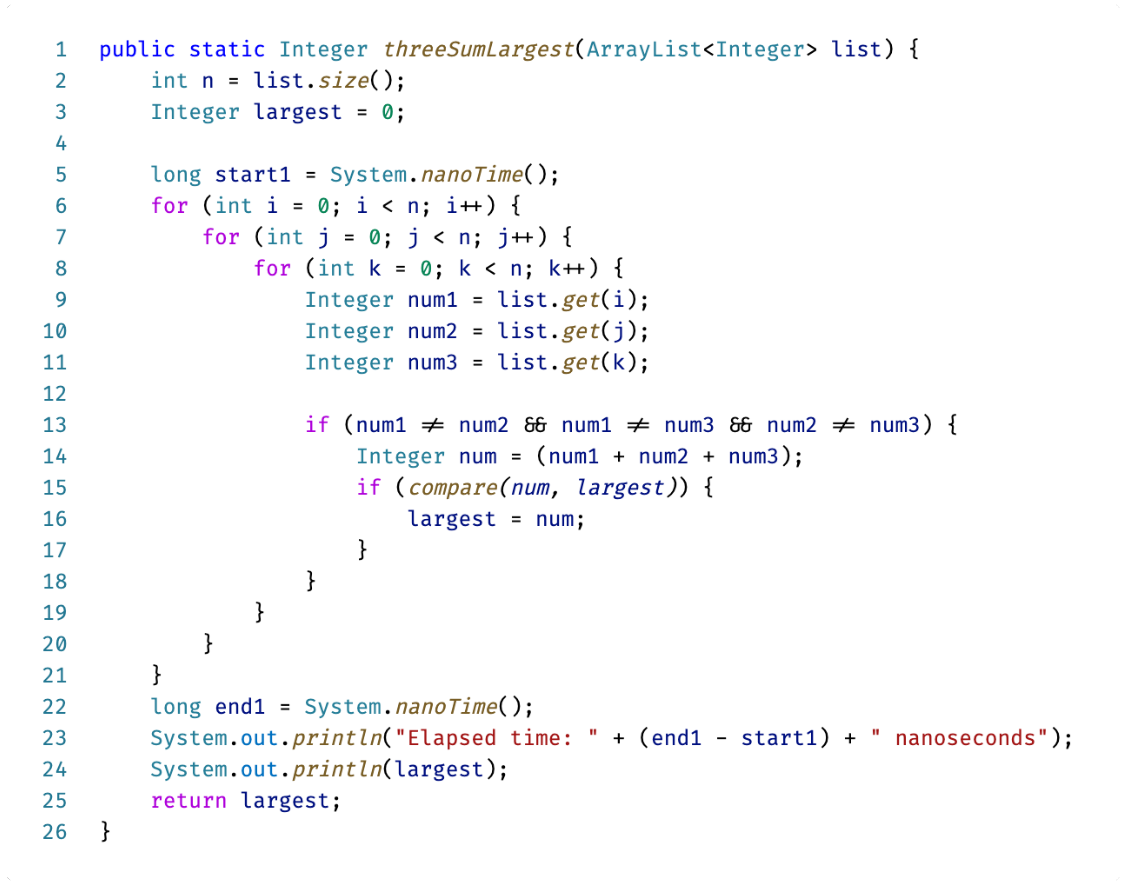
1 Week 1

1.1 Task 1

Given a list of integers, write an algorithm that selects the three numbers that summed gives the largest value. $[8, 19, -1, 27, 3, 9, -40]$. The largest value that three of the numbers sums to is 55. What runtime does your algorithm have?

1.1.1 Solution

I came up with two solutions for this problem, the first problem is a rather slow solution, and as there are three nested for loops, this program runs in $O\left(n^3\right)$ time.



```java
public static Integer threeSumLargest(ArrayList<Integer> list) {
    int n = list.size();
    Integer largest = 0;

    long start1 = System.nanoTime();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                Integer num1 = list.get(i);
                Integer num2 = list.get(j);
                Integer num3 = list.get(k);

                if (num1 != num2 && num1 != num3 && num2 != num3) {
                    Integer num = (num1 + num2 + num3);
                    if (compare(num, largest)) {
                        largest = num;
                    }
                }
            }
        }
    }
    long end1 = System.nanoTime();
    System.out.println("Elapsed time: " + (end1 - start1) + " nanoseconds");
    System.out.println(largest);
    return largest;
}
```

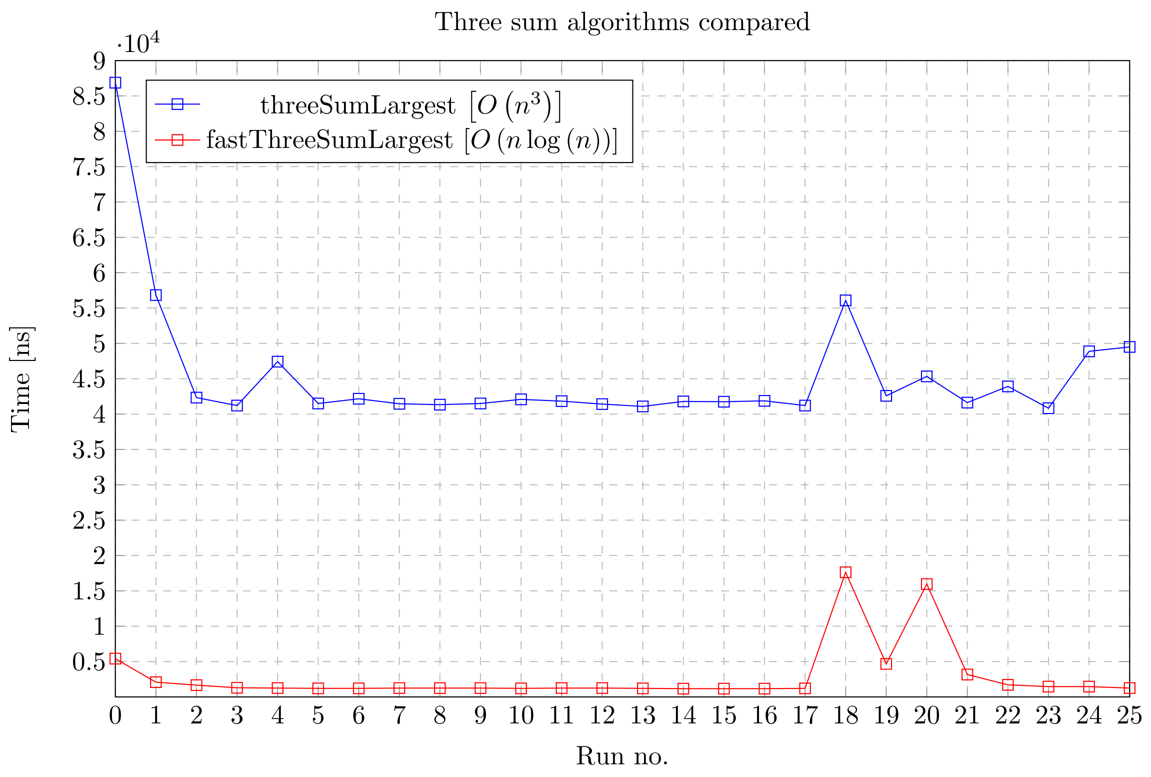
Figure 1: First solution for this problem

The second solution is slightly faster, however finding its time complexity is slightly harder, as we are using some methods from the *Collections* framework. As shown in Figure 2, we can see that *Collections.Sort* and has been used. Doing a quick google search, we find that this method has a time complexity of $O\left(n \log\left(n\right)\right)$.

```java
public static Integer fastThreeSumLargest(ArrayList<Integer> list) {
    long start2 = System.nanoTime();
    int size = list.size();
    Collections.sort(list);

    Integer largest = list.get(size-1) + list.get(size-2) + list.get(size-3);
    long end2 = System.nanoTime();

    System.out.println("Elapsed time: " + (end2 - start2) + " nanoseconds");
    System.out.println(largest);
    return largest;
}
```

Figure 2: Second solution, slightly faster

Comparing the two solutions, we get the the following graph. As we can see, the faster solution is vastly surperior to the slower version.



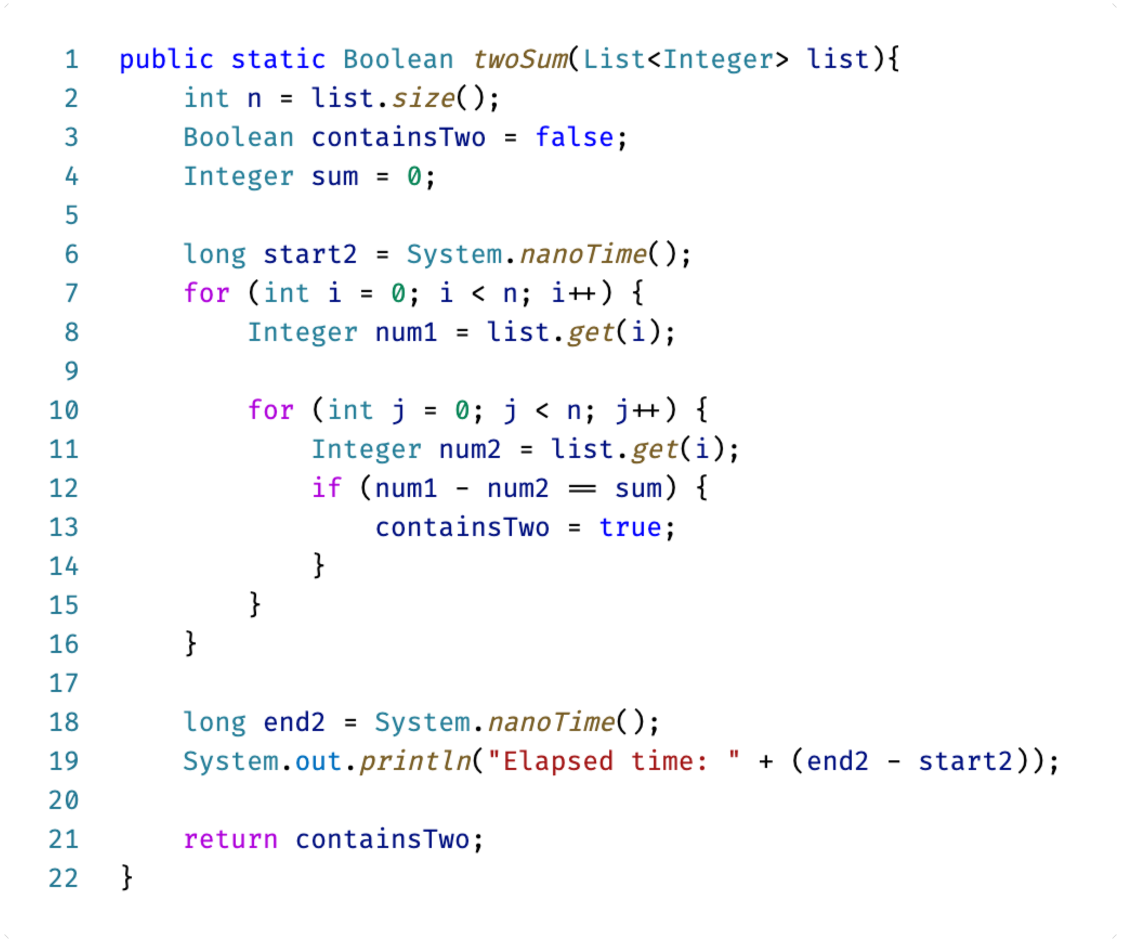
1.2 Task 2

Given a sorted list of integers, write an algorithm that runs in linear time ($O(n)$) that determines if two numbers sum to 0. $[-10, -8, -4, -1, 5, 8, 18, 20, 40]$. mk and 8 sum to 0 hence this list should yield *true*.

1.2.1 Solution

This problem is a *two sum* problem, which usually is solved with a time complexity of $O\left(n^2\right)$. Again two solutions were developed, one of which is the standard solution, running in $O\left(n^2\right)$ time, and the other being a faster solution, running in $O\left(n\right)$ time, or linear time.

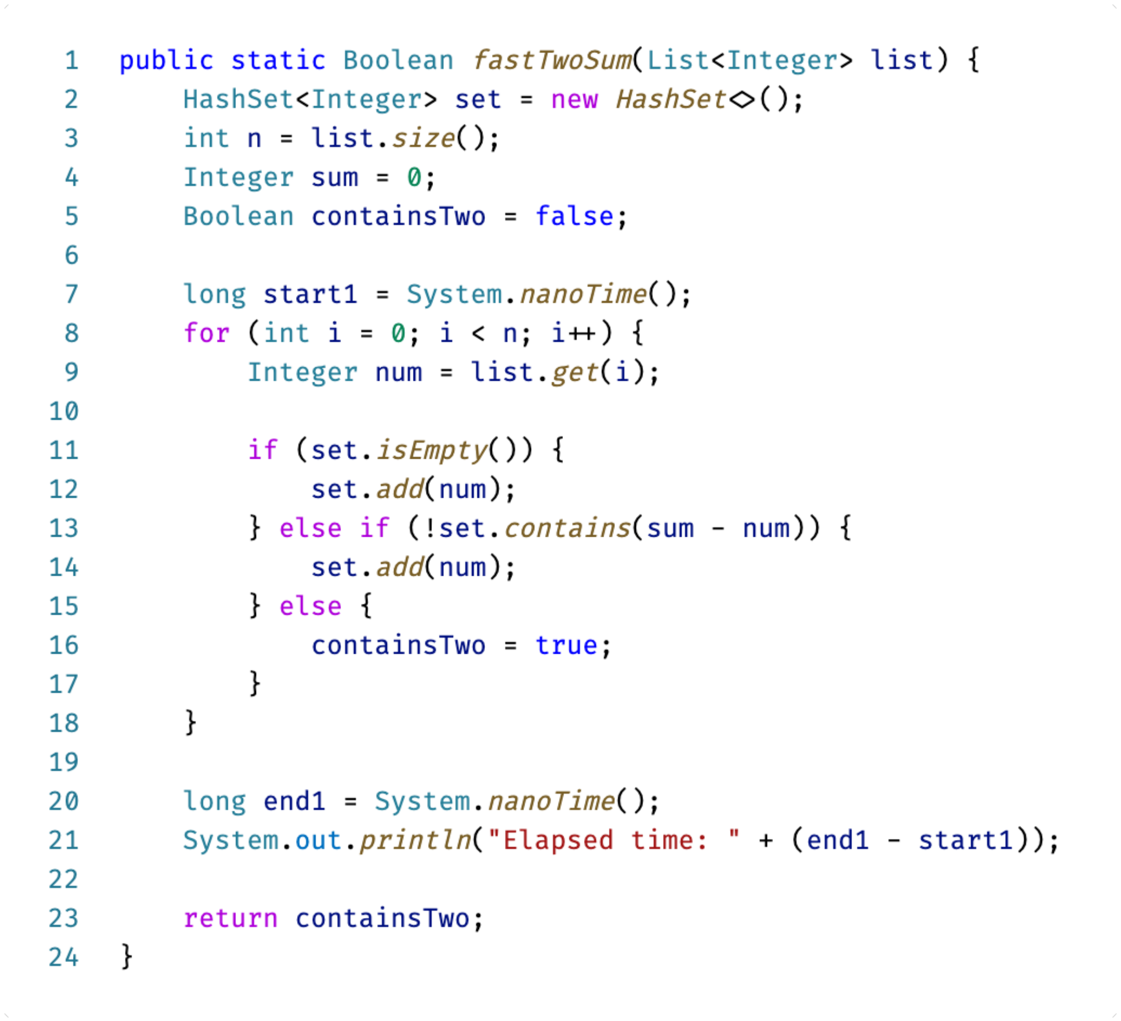
The first solution uses two nested for loops to check if two of the integers sum to the desired goal sum. Because there are two for loops, its a rather slow solution, and it runs in $O\left(n^2\right)$ time, as mentioned.



```java
public static Boolean twoSum(List<Integer> list){
    int n = list.size();
    Boolean containsTwo = false;
    Integer sum = 0;

    long start2 = System.nanoTime();
    for (int i = 0; i < n; i++) {
        Integer num1 = list.get(i);

        for (int j = 0; j < n; j++) {
            Integer num2 = list.get(i);
            if (num1 - num2 == sum) {
                containsTwo = true;
            }
        }
    }

    long end2 = System.nanoTime();
    System.out.println("Elapsed time: " + (end2 - start2));

    return containsTwo;
}
```

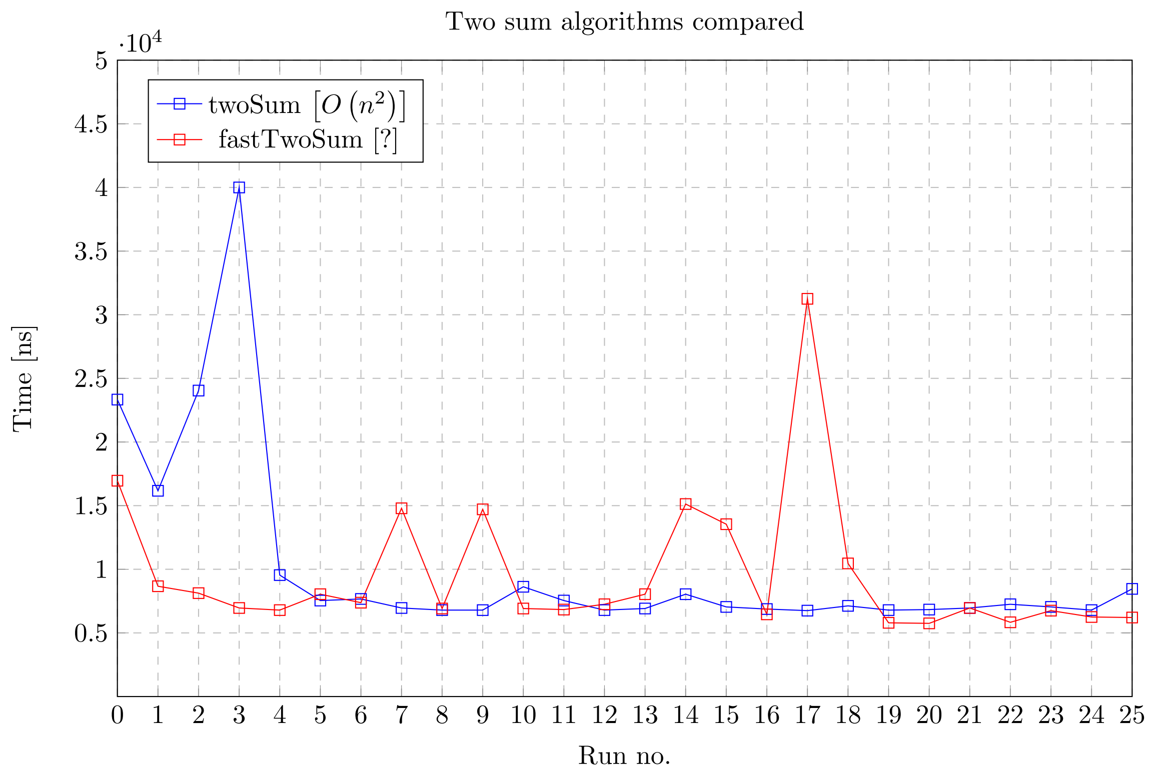
Figure 3: First and slower solution of the problem

The faster solutions, employs the usage of an empty HashSet. It then loops through the list and checks if one of three conditions are met. The first condition occurs when the set is empty, is the simply adds the currenet element to the list. Next it checks if the set *doesn't* contain the result of the sum subtracted from the current element, if it doesn't, it adds the current element to the list. Lastly, if none of the previous conditions are met, it means that we have found that two integers in the list sum to our desired sum, and the variable *containsTwo* is set to true.



```java
public static Boolean fastTwoSum(List<Integer> list) {
    HashSet<Integer> set = new HashSet<>();
    int n = list.size();
    Integer sum = 0;
    Boolean containsTwo = false;

    long start1 = System.nanoTime();
    for (int i = 0; i < n; i++) {
        Integer num = list.get(i);

        if (set.isEmpty()) {
            set.add(num);
        } else if (!set.contains(sum - num)) {
            set.add(num);
        } else {
            containsTwo = true;
        }
    }

    long end1 = System.nanoTime();
    System.out.println("Elapsed time: " + (end1 - start1));

    return containsTwo;
}
```

Figure 4: Second, faster solution



As we can see, it does not look like the ”faster” version of this algorithm actually is faster, further optimization is certainly appropriate.

1.3 Task 3

Sort the functions based on which grows fastest when n becomes large:

- $f(n) = n^2 + n$
- $f(n) = n * \log(n)$
- $f(n) = n \cdot n$
- $f(n) = n \cdot (log(n))^3$

The functions above grow fastest according to the following list

1. $f(n) = n^2 + n$
2. $f(n) = n \cdot n$
3. $f(n) = n \cdot (log(n))^3$
4. $f(n) = n * \log(n)$